2020 자율주행 교육

WeGo 위고 주식회사

목차

WeGo

01

Neural Network Training

- 신경망에서의 기울기

→ 신경망 학습에서도 기울기를 통해 경사 하강법을 적용

→ 신경망 학습에서의 기울기는 가중치에 대한 손실 함수의 기울기를 의미

$$\boldsymbol{W} = \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{pmatrix} \qquad \frac{\partial L}{\partial \boldsymbol{W}} = \begin{pmatrix} \frac{\partial L}{\partial w_{11}} & \frac{\partial L}{\partial w_{12}} & \frac{\partial L}{\partial w_{13}} \\ \frac{\partial L}{\partial w_{21}} & \frac{\partial L}{\partial w_{22}} & \frac{\partial L}{\partial w_{23}} \end{pmatrix}$$

```
class simpleNet:
    def __init__(self):
        self.W = np.random.randn(2, 3)
    def predict(self, x):
        return np.dot(x, self.W)
    def loss(self, x, t):
        z = self.predict(x)
        y = softmax(z)
        loss = cross_entropy_error(y, t)
        return loss
```

01. Neural Network Training

- Random한 2 x 3의 Network 생성 → net = simpleNet()
- 가중치 매개변수 출력 → print(net.W)
- 1 x 2의 Input 생성 → x = np.array([0.6, 0.9])
- Prediction 진행 → p = net.predict(x)
- Prediction 결과 출력 → print(p)
- 결과가 최대가 되는 index 출력 → np.argmax(p)
- 정답 Label 임의 생성 → t = np.array([0, 0, 1])
- Loss function 확인 → net.loss(x, t)
- Gradient 계산 → dW = numerical_gradient(lambda w: net.loss(x, t), net.W)

WeGo

01. Neural Network Training

- Stochastic Gradient Descent
- 미니 배치 (Training Data 중 일부만 Random으로 추출)를 이용하여 loss function의 값을 줄이는 것이 목표
- 각 가중치 매개변수의 기울기를 산출(Gradient는 손실 함수를 줄이는 방향을 제시)
- 가중치 매개변수를 기울기 방향으로 약간 갱신
- 이후 위의 과정을 반복

```
class TwoLayerNet:
    def __init__(self, input_size, hidden_size, output_size, weight_init_std=0.01):
        self.params = {}
        self.params['W1'] = weight_init_std * np.random.randn(input_size, hidden_size)
        self.params['b1'] = np.zeros(hidden_size)
        self.params['W2'] = weight_init_std * np.random.rand(hidden_size, output_size)
        self.params['b2'] = np.zeros(output_size)
    def predict(self, x):
        W1, W2 = self.params['W1'], self.params['W2']
        b1, b2 = self.params['b1'], self.params['b2']
        a1 = np.dot(x, W1) + b1
        z1 = sigmoid(a1)
        a2 = np.dot(z1, W2) + b2
        y = softmax(a2)
        return y
```

```
class TwoLayerNet:
    def loss(self, x, t):
        y = self.predict(x)

        return cross_entropy_error(y, t)

    def accuracy(self, x, t):
        y = self.predict(x)
        y = np.argmax(y, axis=1)
        t = np.argmax(t, axis=1)
        accuracy = np.sum(y == t) / float(x.shape[0])

        return accuracy

    def numerical_gradient(self, x, t):
        loss_W = lambda W: self.loss(x, t)

        grads = {}
        grads['W1'] = numerical_gradient(loss_W, self.params['W1'])
        grads['b1'] = numerical_gradient(loss_W, self.params['b1'])
        grads['W2'] = numerical_gradient(loss_W, self.params['W2'])
        grads['b2'] = numerical_gradient(loss_W, self.params['b2'])

        return grads
```

```
class TwoLayerNet:
    def loss(self, x, t):
        y = self.predict(x)

        return cross_entropy_error(y, t)

    def accuracy(self, x, t):
        y = self.predict(x)
        y = np.argmax(y, axis=1)
        t = np.argmax(t, axis=1)
        accuracy = np.sum(y == t) / float(x.shape[0])

        return accuracy

    def numerical_gradient(self, x, t):
        loss_W = lambda W: self.loss(x, t)

        grads = {}
        grads['W1'] = numerical_gradient(loss_W, self.params['W1'])
        grads['b1'] = numerical_gradient(loss_W, self.params['b1'])
        grads['W2'] = numerical_gradient(loss_W, self.params['W2'])
        grads['b2'] = numerical_gradient(loss_W, self.params['b2'])

        return grads
```

01. Neural Network Training

#MNIST Data 학습

```
(x_train, t_train), (x_test, t_test) = load_mnist(normalize=True, one_hot_label=True)
network = TwoLayerNet(input_size=784, hidden_size=50, output_size=10)

# 하이퍼파라미터
iters_num = 10000  # 반복 횟수를 적절히 설정한다.
train_size = x_train.shape[0]
batch_size = 100   # 미니배치 크기
learning_rate = 0.1

train_loss_list = []
```

01. Neural Network Training

```
#MNIST Data 학습
for i in range(iters_num):
    # 미니배치 획득
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    t_batch = t_train[batch_mask]
    # 기울기 계산
    #grad = network.numerical_gradient(x_batch, t_batch)
    grad = network.gradient(x_batch, t_batch)
    # 매개변수 갱신
    for key in ('W1', 'b1', 'W2', 'b2'):
        network.params[key] -= learning_rate * grad[key]
    # 학습 경과 기록
    loss = network.loss(x_batch, t_batch)
    train_loss_list.append(loss)
```

```
#시험 데이터를 이용하여 평가
(x_train, t_train), (x_test, t_test) = load_mnist(normalize=True, one_hot_label=True)
network = TwoLayerNet(input_size=784, hidden_size=50, output_size=10)

# 하이퍼파라미터
iters_num = 10000  # 반복 횟수를 적절히 설정한다.
train_size = x_train.shape[0]
batch_size = 100   # 미니배치 크기
learning_rate = 0.1

train_loss_list = []
train_acc_list = []
test_acc_list = []


# 1에폭당 반복 수
iter_per_epoch = max(train_size / batch_size, 1)
```

```
#시험 데이터를 이용하여 평가
for i in range(iters_num):
    # 미니배치 획득
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    t_batch = t_train[batch_mask]
    # 기울기 계산
    #grad = network.numerical_gradient(x_batch, t_batch)
    grad = network.gradient(x_batch, t_batch)
    # 매개변수 갱신
    for key in ('W1', 'b1', 'W2', 'b2'):
        network.params[key] -= learning_rate * grad[key]
    # 학습 경과 기록
    loss = network.loss(x_batch, t_batch)
    train_loss_list.append(loss)
    # 1에폭당 정확도 계산
    if i % iter_per_epoch == 0:
        train_acc = network.accuracy(x_train, t_train)
        test_acc = network.accuracy(x_test, t_test)
        train_acc_list.append(train_acc)
        test_acc_list.append(test_acc)
        print("train acc, test acc | " + str(train_acc) + ", " + str(test_acc))
```

```
#시험 데이터를 이용하여 평가
# 그래프 그리기
markers = {'train': 'o', 'test': 's'}
x = np.arange(len(train_acc_list))
plt.plot(x, train_acc_list, label='train acc')
plt.plot(x, test_acc_list, label='test acc', linestyle='--')
plt.xlabel("epochs")
plt.ylabel("accuracy")
plt.ylim(0, 1.0)
plt.legend(loc='lower right')
plt.show()
```

02

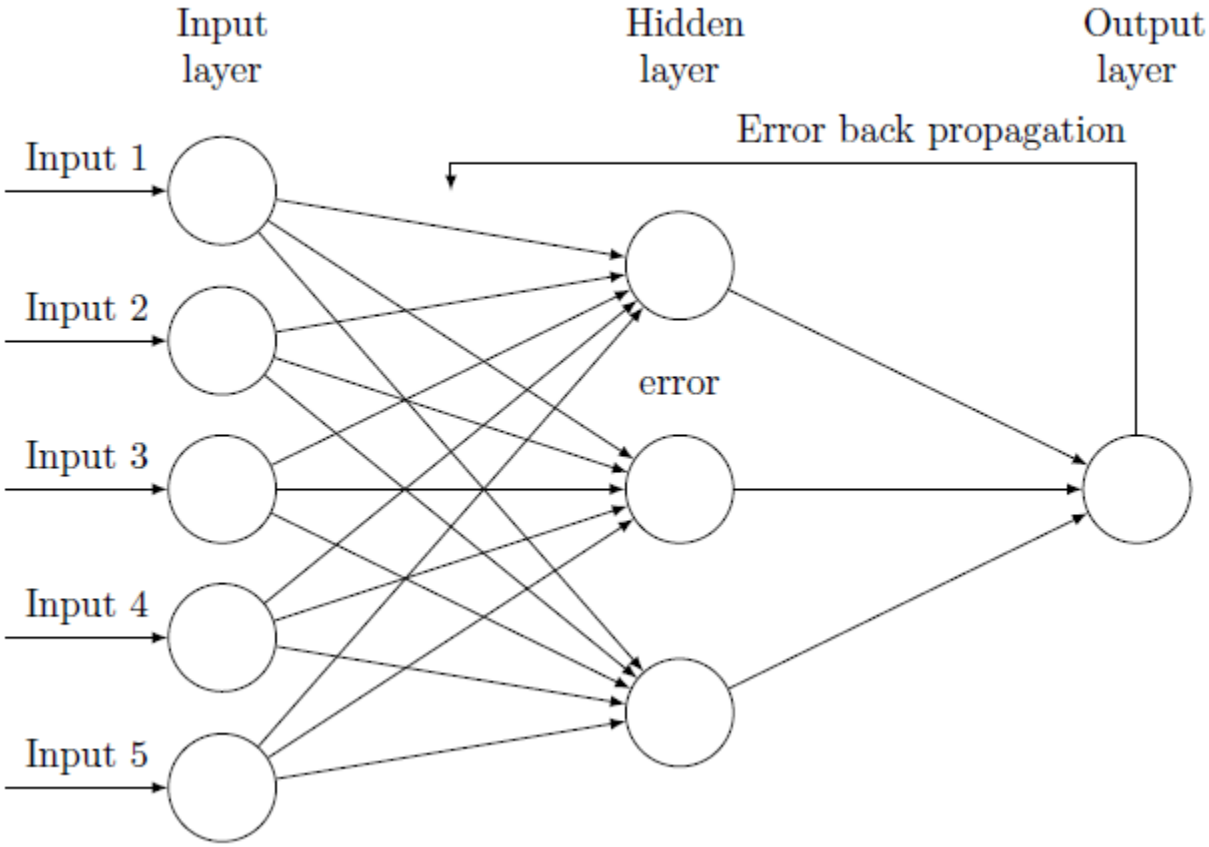
Error Backpropagation

02. Error Backpropagation

- 앞서 신경망 학습에서는 Gradient Descent(경사 하강법)을 이용하여, 가중치 매개변수의 값을 수정
- 이는 이해하기 쉽고, 간단하지만, 시간이 오래 걸린다는 단점
- 이를 보완하기 위해서 등장한 방법이 오차 역전파 방법

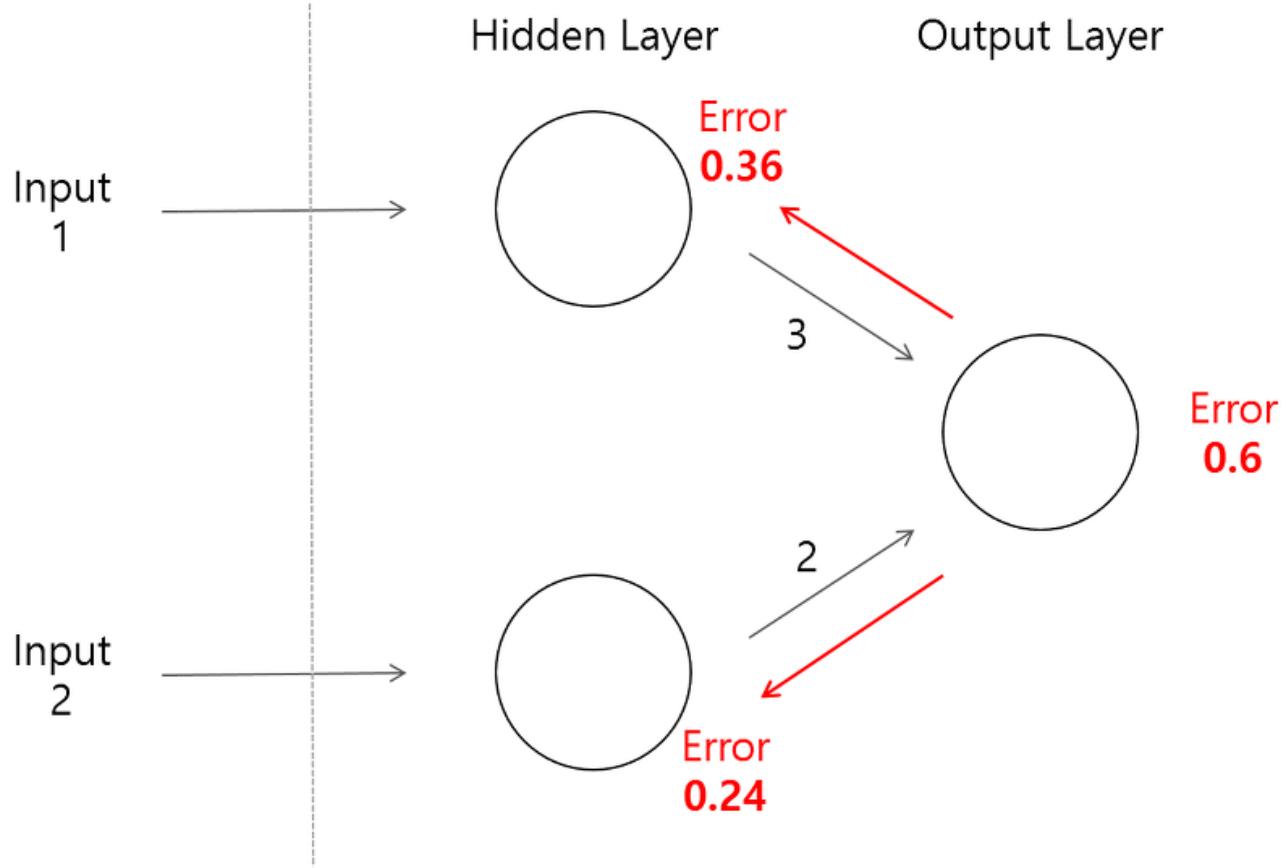
02. Error Backpropagation

- 오차 역전파법
- 순전파와 반대되는 말로, 경사 하강법과 같이 가중치를 업데이트 하는 방법

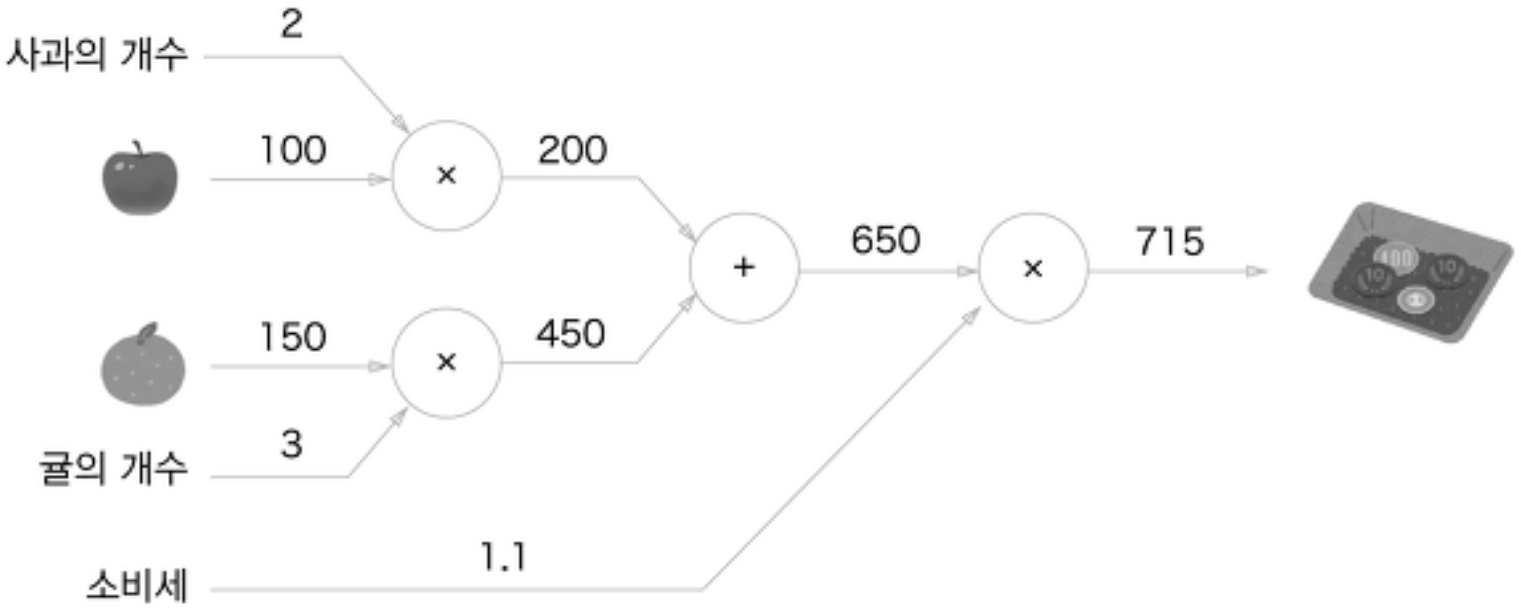


02. Error Backpropagation

- Error가 발생한 원인에 해당하는 부분을 계산하여 역으로 전달하는 방법
- 입력으로 들어온 값의 크기에 따라서 역전파되는 Error의 크기도 증가

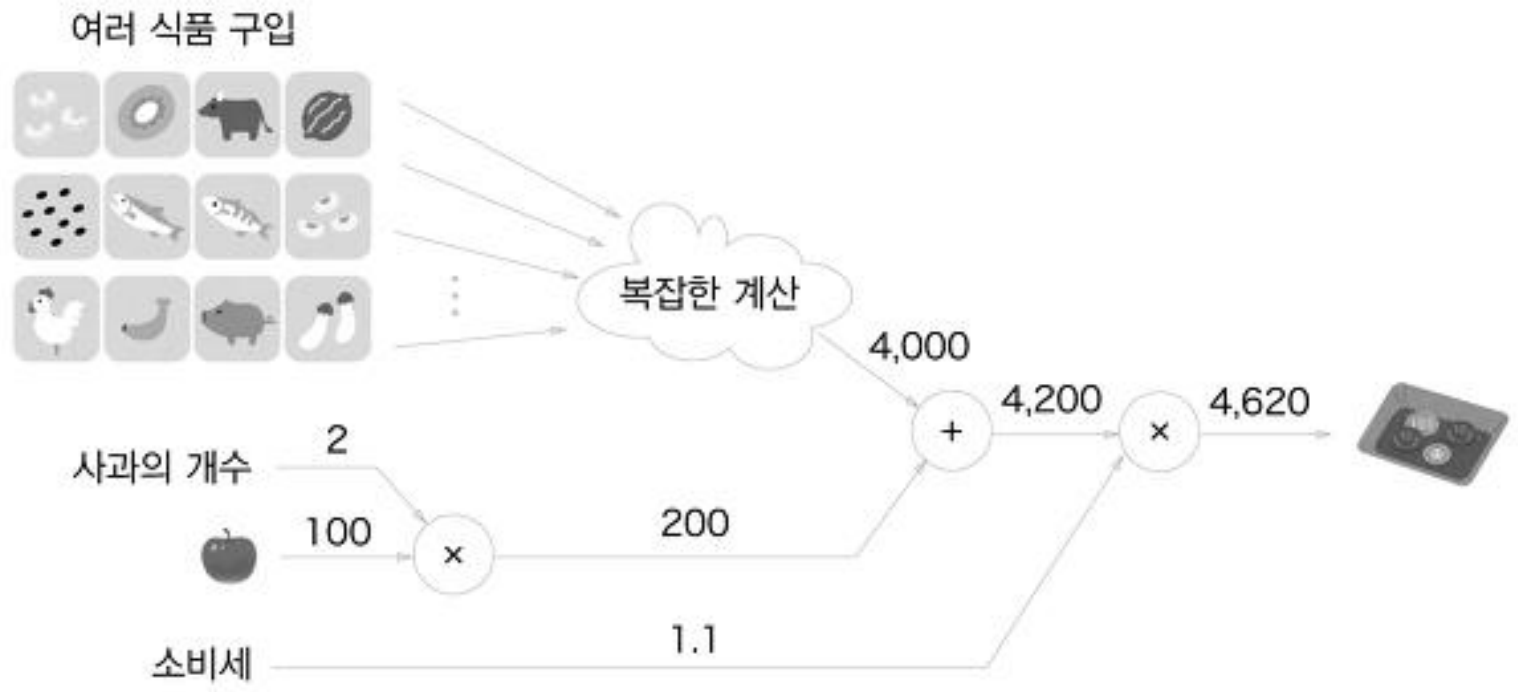


- 계산 그래프를 이용한 방법
- 사과 2개, 귤 3개, 최종적으로 소비세가 10%
- 그래프를 통해 왼쪽에서 오른쪽으로 계산을 진행 (순전파)



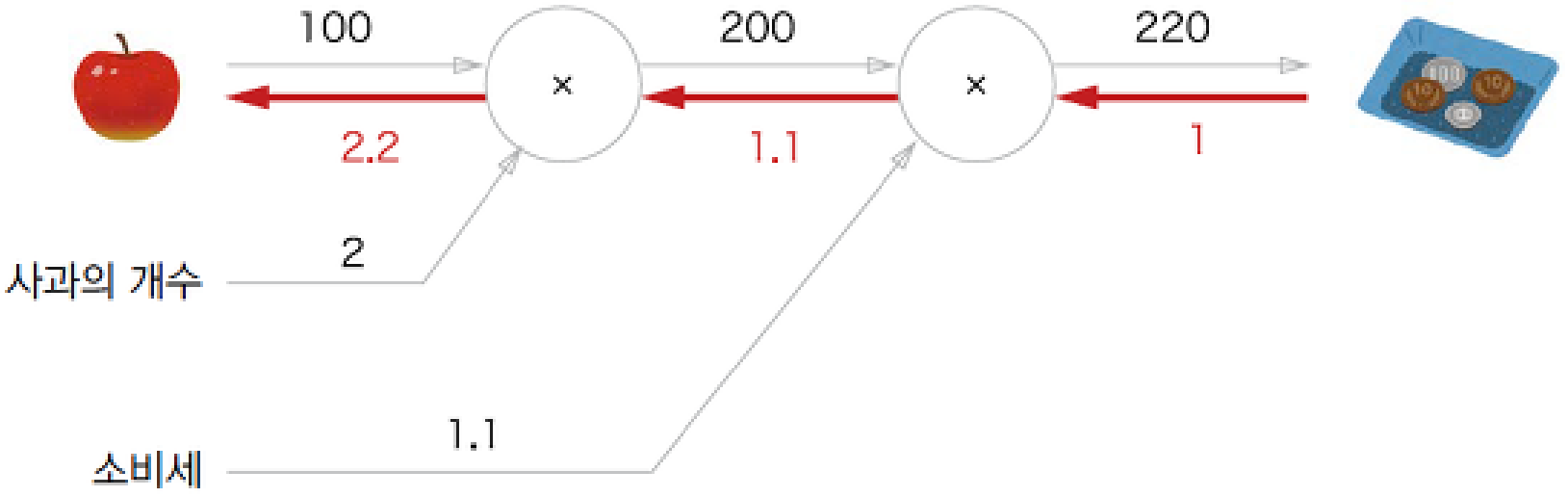
02. Error Backpropagation

- 계산 그래프의 경우, 국소적 계산을 통해 최종 결과 값을 얻을 수 있음
- 국소적 계산 → 자신과 직접 관계된 범위에서만 계산을 진행



02. Error Backpropagation

- 계산 그래프의 장점
- 국소적 계산을 통해, 문제를 단순화 할 수 있음
- 역전파를 통해 미분값을 효율적으로 계산할 수 있음



- 연쇄법칙
- 합성 함수의 미분은 각 함수의 곱으로 계산이 가능

$$t = x + y$$

$$z = t^2$$

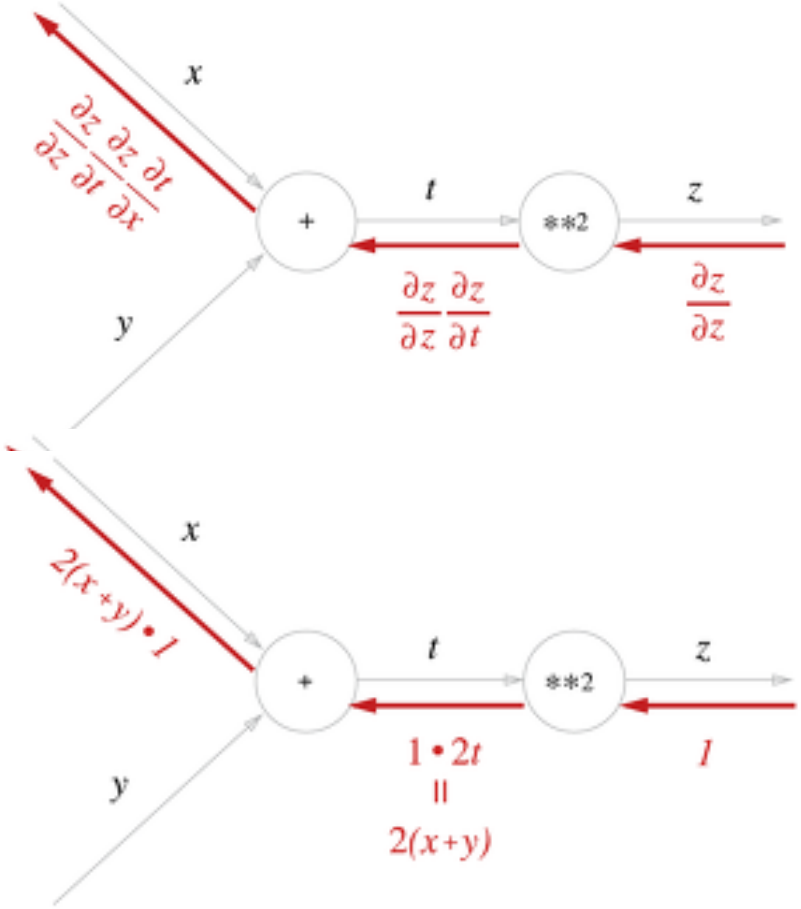
$$\frac{\partial z}{\partial t} = 2t$$

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial t}\frac{\partial t}{\partial x}$$

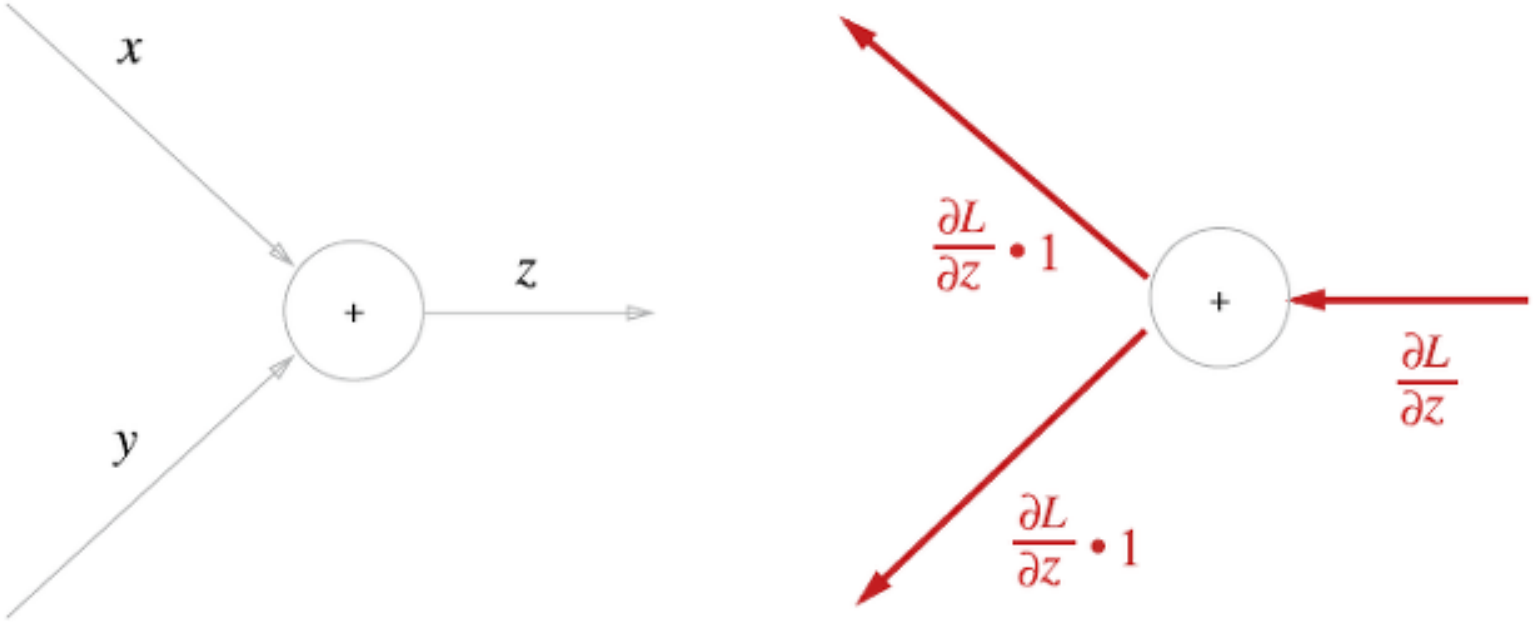
$$\frac{\partial t}{\partial x} = 1$$

$$\frac{\partial z}{\partial x} = 2t = 2(x + y)$$

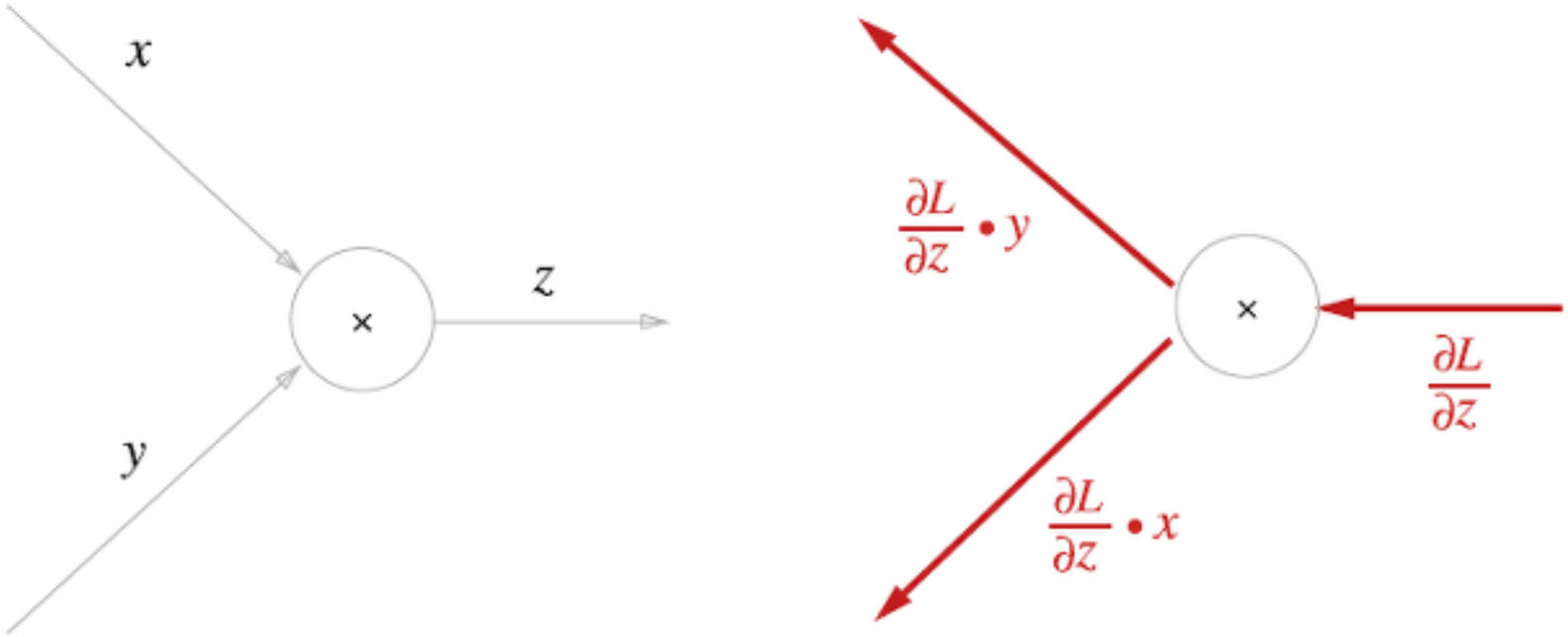
- 연쇄법칙
- 합성 함수의 미분은 각 함수의 곱으로 계산이 가능



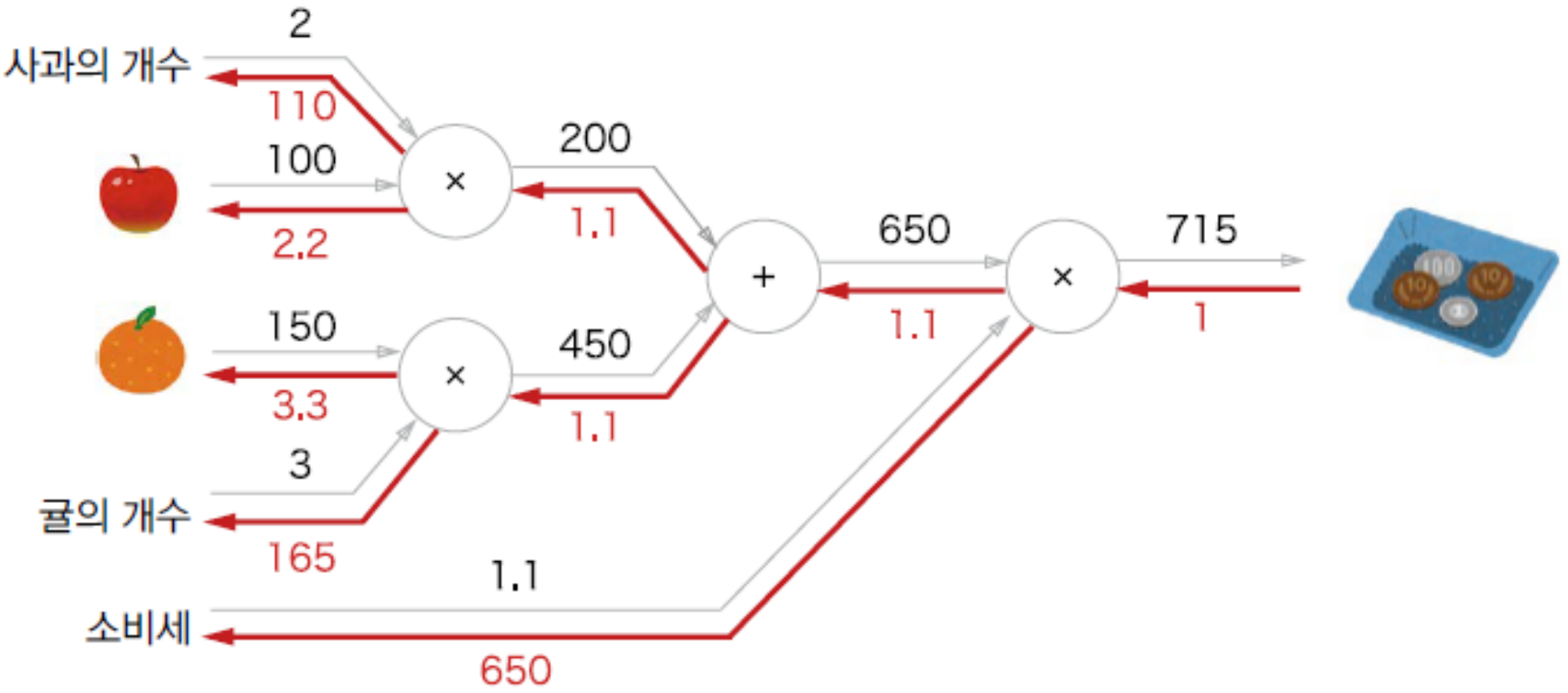
- 덧셈 노드의 역전파
- 덧셈 노드의 경우, 입력값을 그대로 흘려보내게 되므로, Gradient Distributor라고 함



- 곱셈 노드의 역전파
- 곱셈 노드의 경우, 입력값의 위치를 서로 바꾸어 곱한 후, 흘려보내므로, Gradient Switcher라고 함



- 계층 구현


2
사과의 개수
110
100
200
×
1.1
2.2
650
715
+
×
1.1
1
150
450
×
3.3
1.1
3
귤의 개수
165
1.1
소비세
650

```
class MulLayer:
    def __init__(self):
        self.x = None
        self.y = None
    def forward(self, x, y):
        self.x = x
        self.y = y
        out = x * y
        return out
    def backward(self, dout):
        dx = dout * self.y  # x와 y를 바꾼다.
        dy = dout * self.x
        return dx, dy
class AddLayer:
    def __init__(self):
        pass
    def forward(self, x, y):
        out = x + y
        return out
    def backward(self, dout):
        dx = dout * 1
        dy = dout * 1
        return dx, dy
```

```
apple = 100 apple_num = 2 orange = 150 orange_num = 3 tax = 1.1
# layer
mul_apple_layer = MulLayer()
mul_orange_layer = MulLayer()
add_apple_orange_layer = AddLayer()
mul_tax_layer = MulLayer()
# forward
apple_price = mul_apple_layer.forward(apple, apple_num)  # (1)
orange_price = mul_orange_layer.forward(orange, orange_num)  # (2)
all_price = add_apple_orange_layer.forward(apple_price, orange_price)  # (3)
price = mul_tax_layer.forward(all_price, tax)  # (4)
# backward
dprice = 1
dall_price, dtax = mul_tax_layer.backward(dprice)  # (4)
dapple_price, dorange_price = add_apple_orange_layer.backward(dall_price)  # (3)
dorange, dorange_num = mul_orange_layer.backward(dorange_price)  # (2)
dapple, dapple_num = mul_apple_layer.backward(dapple_price)  # (1)
print("price:", int(price))
print("dApple:", dapple)
print("dApple_num:", int(dapple_num))
print("dOrange:", dorange)
print("dOrange_num:", int(dorange_num))
print("dTax:", dtax)
```

WeGo